Intro to Computer Systems

Assessment 2

1.

A) \*I am interpreting the ‘description of an algorithm’ as a pseudocode equivalent. I am also providing a written description in English of the given pseudocode

Algorithm: Produce a 4x4 space invaders on a Brookshear Machine Display.

Input: User Running program.

Output: Correct display of space invader figures.

false = 00;

Required register values: 0 - a comparator to allow jumps; 1 – incrementor; 80 – store display location; 4 – mark end of row, hence a register to start at 0 and increment to 3. Two Final registers will be required: one to store the pattern to be displayed, and another to store the result of logical operations.

increment = 01;

location = 80;

rowMark = 04;

condition = 00;

pattern1 = 01111100;

DATA values will also be required to store the patterns that will be displayed. There are 7 varying patterns that need to be implemented on each row. Once the final pattern has been displayed, the next row will need to display the first pattern again.

pattern2= 11010110;

pattern3 = 11111110;

pattern4 = 00111000;

pattern5 = 01010100;

pattern6 = 10000010;

pattern7 = 00000000;

First loop will terminate when final display location, which is 00, has been reached.

Inner loop will terminate when pattern1 has been displayed 4 times. Once it has been displayed 4 times, the end of the first row will have been reached, meaning our condition variable will have to be reset and a new loop for pattern2 will begin.

while location != 00 do

while condition < rowMark do

[location] = pattern1;

location++;

condition++;

End while

Condition = 00;

while condition < rowMark do

Same as the inner loop above, it will terminate when pattern2 has been displayed 4 times.

This will then lead to the next loop responsible for displaying pattern3.

[location] = pattern2;

location++;

condition++;

End while

condition = 00;

while condition < rowMark do

Same as above, but displaying pattern3.

Condition, which goes from 00 to 03, is reset each time.

Location is incremented by 1. This allocates the next storage location to display the pattern.

[location] = pattern3;

location++;

condition++;

End while

condition = 00;

while condition < rowMark do

[location] = pattern4;

Same as above, but displaying pattern4.

location++;

condition++;

End while

condition = 00;

while condition < rowMark do

Same as above, but displaying pattern5.

[location] = pattern5;

location++;

condition++;

End while

condition = 00;

while condition < rowMark do

Same as above, but displaying pattern6.

[location] = pattern6;

location++;

condition++;

End while

condition = 00;

while condition < rowMark do

Same as above, but displaying pattern7.

However, on incrementation of location, loop will go back to the beginning and check whether location is equal to 00 (or has exceeded FF).

If it hasn’t, the entire loop will repeat, creating a duplication of the previous 7 rows (which should display 4 space-invaders). This loop should repeat 4 times (plus another 4 rows to create an incomplete space invader).

If it has exceeded location FF, program will Halt .

[location] = pattern7;

location++;

condition++;

End while

condition = 00;

End while

End Algorithm

B)

See ‘Program.txt’ file.

C)

When the program executes, it outputs a display identical to one provided in the assignment brief. In total, 730 instructions executed.

Initially, when considering solutions to this problem, there where two approaches I could have taken: One would be by creating a main loop function. This entire loop would be responsible for creating 4 complete space invader characters, by checking when the Location registry had been incremented 28 times (producing 4 complete characters). The problem with this approach was that the last row did not produce 4 complete space invaders thus meaning the loop would finish incomplete. The second approach, and the one I choose, was to create individual loops for each row of 4 bytes, that would execute 4 times before jumping to the next loop. Although the first approach would have produced a more efficient code, I decided to stick with this approach as I felt it would be more professional to have 7 complete loops for each row, as opposed to 1 main loop which was responsible for creating each row of 4 characters, which would end with half the loop being executed.

The program initially assigns registers with relevant data variables.

It then creates a wrapping loop where the rest of the program is contained. This wrapping loop will run until the register containing the location exceeds the final memory location, which is FF.

Within the wrapping loop, there are 7 inner loops responsible for displayed each row of the space invader character. As the display is 32x32(bits), and each character is comprised of 8x7(bits), each individual loop must run 4 times. This is done using ‘JMPEQ’ and ‘JMP’ commands (which will be further discussed in part ‘D)’).

Once each of the 7 inner loops have been executed 4 times, the first 4 characters will have been created. The program will then need to go back to the start of the wrapping loop, and begin the next set of space invader characters.

This will carry on until the register responsible for storing the location of where to print on the display exceeds FF. Once FF has been exceeded, the wrapping loop will terminate and jump to a halt instruction.

This will leave a display resembling the one from the assignment spec.

D)

When the program is run, it assigns 6 main registers: R0 – Condition comparator (Used for JMPEQ tests to check whether AND operations return false. R1 – Incrementor (Set to 1 and added to other registers). R2 – Location (Stores location of where to print the provided display). R3 – Row count (Stores 04 which is used to check when a loop has been executed 4 times). R4 – conditional (Stores the result of an AND operation between registers 3 and 6, which shows when the end of a row has been reached). R5 – bit Pattern (The pattern that should be displayed across a row). R6 – row count (Increments up to 4 to mark position of row).

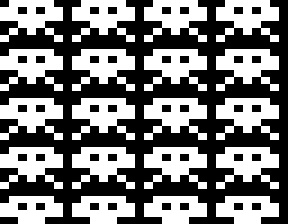
There are also 7 DATA variables named ‘pattern\_’. These are commented in the code, and store each individual bit pattern responsible for each row of a space invader. On attempt of using these, the register seems to only store the first 4 bits. This faults the display, and I therefore had to resort to moving the binary equivalent of the register manually. This is commented in the ‘Program.txt’.

‘loop01’ marks the start of a row by setting R6 as 00. It then goes to ‘loop’, which is responsible for displaying the first bit pattern 4 times across the top row. It does this by moving ‘pattern1’ into R5, which is then indirectly addressed to the memory address of R2. We then increment both R2 (to get to the next location) and R6 (to keep a record on how far along the row we have traversed). An AND operation is then preformed between R6 (which now holds 1) & R3 (which holds 4). This value is then stored in R4. If the returned value is 01, we have incremented R6 4 times, and therefore have reached the end of the row, meaning it’s time to go to the next loop. However, if the returned result is 00 (which it will be for the first execution), then we know we have not reached the end of the row, and we still need to continue with the current bit pattern stored in R5. Therefore, a JMP back to ‘loop’ is made, and the process above is repeated, only without resetting R6.

When the loop has executed 4 times, and the return of the AND operation is 01, we can move onto ‘loop21’. ‘loop21’ is a near recreation of ‘loop’, with only 2difference: A new bit pattern (‘pattern2’) in R5, and instead of called back to ‘loop’ or ‘loop21’, we make arguments to call to ‘loop2’ or ‘loop31’ (Therefore this version of the code is slightly more inefficient than the approach mentioned in ‘C)’: using repeat instructions, but produces a more complete and understandable code, still within the given requirements). This loop will perform in the same fashion, as so will loops ‘loop3’ to ‘loop7’.

The only two differences are loops ‘loop4’ and ‘loop7’. ‘loop4’ contains a test which will make a call to ‘halt’ when register R2 is equal to 00. This signifies the All the memory locations responsible for the display have been written too, and therefore the display is complete (Taking my first approach from ‘C)’, this could be incorporated in a main loop to make the code more efficient. However, I didn’t like the idea of including this test midway through a loop, so I choose this approach to solve the problem). Instead of ‘loop7’ making further calls, it calls back to ‘loop’ as ‘loop7’ signifies the last row needed to produce 4 complete space invader characters.

When ‘loop4’ does make a call to ‘halt’, the halt instruction will execute and the program will stop, leaving a display representing the one provided in the spec.



2.

A)

A disassembler is the opposite to an assembler: it converts machine code into assembly code. This is particularly useful when analysing existing code that needs to be modified.

Although it is possible to analyse a machine code given its instruction set, the advantage of using a disassembler is to provide clarity on variables which are assigned by the program, as opposed to variables which are automatically assigned by the hardware.

However, there are disadvantages of disassemblers. If the original source code of a program is written using constants and specific comments to provide clarity on its performance, these are not returned when using a disassembler. This often leads to assembly code which is more appropriate for a human as opposed to an assembler. There is also an issue when disassembling code which involves jump instructions. A disassembler cannot distinguish the difference between an absolute jump instruction: one leading to specific code to be executed, as opposed to a relative jump instruction: One intended to avoid specific code. This causes an issue if the programmer wishes to add code between the jump command.

Disassemblers can be useful in detecting malware (static malware analysis). This involves disassembling malicious code before it has been executed. This means the code can be understood in a readable format, before it has activated itself. But understanding the code, an analysist can understand its overall intentions as a program, and act accordingly.

However, it is possible that malware machine code can contain syntactical errors in order to elude its true purpose, yet preforming in a malicious manner. This then requires dynamic malware analysis to understand what the code does in runtime.

B)

An interpreter preforms code without the need to compiling it first. Its operates by either: parsing each individual String syntax and executing it, translating the code and storing it in an efficient data structure, or by referencing precompiled code (made by a compiler, which is part of the interpreter) and executing it. Popular languages like Perl and Python use the second method, converting source code into a given data structure. This is essentially a form of compression and intended to better optimisation and translation. However, it is important that by converting the source code into an Intermediate representation, no information from the source code is lost.

The advantage of interpreters is that they can quickly interpret source code into machine code and execute each instruction line by line. This is advantageous over a compiler which translates the entire sources code into a complete machine code representation based on a one-to-one mapping of high level code to machine code.

An example of a language which utilises both a compiler and an interpreter is java. Javac is javas compiler which converts source code directly to machine code. However, this only works for platform specific translations. In order to execute the same java source code on multiple platforms, Java Virtual Machine acts an interpreter, executing each machine code instruction one by one with accordance to the platform it is being run on.

This is highly advantageous as it allows the same code to be executed on many platforms, which is Java’s main commodity.

References:

Disassembler  
<http://www.tech-faq.com/disassembler.html>

<https://en.wikipedia.org/wiki/Malware_analysis>

Interpreter

<https://en.wikipedia.org/wiki/Interpreter_(computing)#Compilers_versus_interpreters>

<https://www.quora.com/Why-does-Java-use-both-compiler-as-well-as-Interpreter>